show

Release 1.4.7

Mar 23, 2017

Contents

`show` provides simple, effective debug printing.

Every language has features to print text, but they're seldom optimized for printing debugging information. `show` is. It provides a simple, DRY mechanism to "show what's going on"–to identify what values are associated with program variables in a neat, labeled fashion. For instance:

```
from show import show

x = 12
nums = list(range(4))

show(x, nums)
```

yields:

```
x: 12  nums: [0, 1, 2, 3]
```

You could of course get the same output with Python's standard `print` statement/function:

```
print("x: {}  nums: {}".format(x, nums))
```

But that's much more verbose, and unlike `show`, it's fails the DRY principle.

But while avoiding a few extra characters of typing and a little extra program complexity is nice–very nice, actually–`show` goes well beyond that. It has methods to show all local variables which have recently changed, to trace the parameters and return values of function calls, and other useful information that you simply cannot get without a lot of needless extra work and a lot of extra lines mucking up your program source. And if you run `show.prettyprint()`, values will be automagically highlighted with Pygments, which is very helpful when you're displaying complex objects, dictionaries and other structures.

As just one exmaple, if you have a lot of output flowing by, and it's hard to see your debugging output, just:

```
show(x, y, z, style='red')
```

And now you have debug output that clearly stands out from the rest. Or `show.set(style='blue')` and now all your debug output is colorized.

But "debug printing is so very 1989!" you may say. "We now have logging, logging, embedded assertions, unit tests, interactive debuggers. We don't need debug printing."

Have to disagree with you there. All those tools are grand, but often the fastest, simplest way to figure out what's going in a program is to print values and watch what happens the program runs. Having a simple, effective way to do that doesn't replacing logging, assertsions, unit testing, and debuggers; it's a effective complement to them. One that is especially useful in two parts of the development process:

1. In exploratory programming, where the values coming back from new or external functions (say, some package's API with which you may not be intimately familiar) aren't well-known to you.
2. In debugging, where the assumptions embedded into the code are clearly, at some level, not being met. Else you wouldn't need to debug.

In either case, knowing what values are actually happening, and figuring them out without a lot of extra effort or complexity–well, it doesn't matter how many unit tests or logging statements you have, that's still of value.

Every language has features to print text, but they're seldom optimized for printing debugging information. `show` is. It provides a simple, DRY mechanism to "show what's going on."

CHAPTER 1

Diving In

Sometimes programs print so that users can see things, and sometimes they print so that developers can. `show()` is for developers, helping rapidly print the current state of variables. A simple invocation:

```
show(x)
```

replaces require the craptastic repetitiveness of:

```
print "x: {0}".format(x)
```

If you'd like to see where the data is being produced,:

```
show.set(where=True)
show(d)
```

will turn on location reporting, such as:

```
__main__():21:   d: 'this'
```

The `where` property, along with most every option, can be set permanently, over the scope of a `where` block, or on a call-by-call basis. `show` is built atop the options module for configuration management, and the output management of say. All `say` options are available. If you `show()` a literal string, it will be interpolated as it would be in `say`:

```
show("{n} iterations, still running")
```

yields something like:

```
14312 iterations, still running
```

But:

```
s = '{n} iterations'
show(s)
```

yields:

```
s: '{n} iterations'
```

See say say for additional detail on its operation. show directly supports many say methods such as blank_lines, hr, sep, and title which are meant to simplify and up-level common formatting tasks.

This Just In

A new capability is to differentially set the formatting parameters on a method by method basis. For example, if you want to see separators in green and function call/return annotations in red:

```
show.sep.set(style='green')
show.inout.set(style='red')
```

You could long do this on a call-by-call basis, but being able to set the defaults just for specific methods allows you to get more formatting in with fewer characters typed. This capability is available on a limited basis: primarily for format-specific calls (blanklines, hr, sep, and title) and for one core inspection call (the inout decorator). It will be extended, and mapped back to underlying say and options features over time.

CHAPTER 2

Collections and Items

The goal of `show` is to provide the most useful information possible, in the quickest and simplest way. Not requiring programmers to explicitly restate values and names in print statements is the start, but it also provides some additional functions that provide a bit more semantic value. For example, `say.items()` is designed to make printing collections easy. It shows not just the values, but also the cardinality (i.e., length) of the collection:

```
nums = list(range(4))
show.items(nums)
```

yields:

```
nums (4 items): [0, 1, 2, 3]
```

CHAPTER 3

Object Properties

```
show.props(x)
```

shows the properties of object `x`. ("Properties" here is generic language for "values" or "attributes" associated with an object, and isn't used in the technical sense of Python properties.) Properties will be listed alphabetically, but with those starting with underscores (_), usually indicating "private" data, sorted after those that are conventionally considered public.

If `x` has real `@property` members, those too displayed. However, other class attributes that `x` rightfully inherits, but that are not directly present in the `x` instance, will not be displayed.

An optional second parameter can determine which properties are shown. E.g.:

```
show.props(x, 'name,age')
```

Or if you prefer the keyword syntax, this is equivalent to:

```
show(x, props='name,age')
```

Or if you'd like all properties except a few:

```
show.props(x, omit='description blurb')
```

CHAPTER 4

Wax On, Wax Off

Often it's convenient to only display debugging information under some conditions, but not others, such as when a `debug` flag is set. That often leads to multi-line conditionals such as:

```
if debug:
    print "x:", x, "y:", y, "z:", z
```

With `show` it's a bit easier. There's a keyword argument, also called `show`, that controls whether anything is shown. If it's truthy, it shows; falsy, ad it doesn't:

```
show(x, y, z, show=debug)
```

You can set the show flag more globally:

```
show.set(show=False)
```

You can also make multiple `show` instances that can be separately controlled:

```
show_verbose = show.clone()
show_verbose.set(show=verbose_flag)
show_verbose(x, y, z)
```

For a more fire-and-forget experience, try setting visibility with a lambda parameter:

```
debug = True
show.set(show=lambda: debug)
```

Then, whenever `debug` is truthy, values will be shown. When `debug` is falsy, values will not be shown.

When you really, truly want `show`'s output to disappear, and want to minimize overhead, but don't want to change your source code (lest you need those debug printing statements again shortly), try:

```
show = noshow
```

This one line will replace the `show` object (and any of its clones) with parallel `NoShow` objects that simply don't do anything or print any output.

Note: This assignment should be done in a global context. If done inside a function, you'll need to add a corresponding `global show` declaration.

As an alternative, you can:

```
from show import show
from show import noshow as show
```

Then comment out the''noshow'' line for debugging, or the `show` line for production runs.

Note: A little care is required to configure global non-showing behavior if you're using `show`'s function decorators such as `@show.inout`. Decorators are evaluated earlier in program execution than the "main flow" of program execution, so it's a good idea to define the lambda or `noshow` control of visibility at the top of your program.

CHAPTER 5

How Things Are Shown

By default, `show` uses Python's `repr()` function to format values. You may prefer some other kind of representation or formatting, however. For example, the `pprint` module pretty-prints data structures. You can set it to be the default formatter:

```
from pprint import pformat
show.set(fmtfunc=pformat)
```

Or to configure separate data and code formatters:

```
show.set(fmtfunc=lambda x: pformat(x, indent=4, width=120, depth=5))
show.set(fmtcode=lambda x: highlight(x, ...))
```

As a convenience, the `show.prettyprint()` configures `pygments` and `pprint` in concert to more attractively display text on ANSI terminals. Just run it once after importing `show`. It also takes `indent`, `depth`, and `width` options for `pformat` and the `style` (style name) option for `pygments`. Some style names to try:

```
# monokai manni rrt perldoc borland colorful default
# murphy vs trac tango fruity autumn bw emacs vim pastie
# friendly native
```

CHAPTER 6

Where Am I?

In addition to the `where` parameter that may be turned on for each `show` call (or in general), there is a method:

```
show.where()
```

intended to display a "flag" or indicator in output where particular debugging output originated. You may not want location on all the time, but an occasional oriented signpost can help.

CHAPTER 7

What's Changed

```
show.changed()
```

will display the value of local variables. When invoked again, only those variables that have changed (since the last `show.changed()` in the same context) will be displayed. For example:

```
def f():
    x = 4
    show.changed()
    x += 1
    retval = x * 3
    show.changed()
    return retval
```

When run will display:

```
x: 4
x: 5  retval: 15
```

You may `omit` some local variables if you like. By default, those starting with underscores (_) will be omitted, as will those containing functions, methods, builtins, and other parts Python program infrastructure. If you'd like to add those, or global variables into the mix, that's easily done:

```
show.changed(_private, MY_GLOBAL_VAR)
```

Will start watching those.

CHAPTER 8

Function Call and Return

It's often helpful to know what a function's parameters and corresponding return values were, and it can be annoying to manually print them out. No matter. Show has two decorators to make this easy:

```
@show.inout
def g(a):
    b = 3
    a += b
    return a

g(4)
```

Displays:

```
g(a=4)
g(a=4) -> 7
```

The first line indicates the function being called. Additional debugging or program output may follow it. The second line here is displayed when the function returns. It reminds us what the parameters were, and then shows what return value resulted. If you like, you can specify the styling of these calls, e.g. with `@show.inout(style='red')`.

While printing both the call entry and exit is often helpful, especially if many lines of output (or potential program crashes) may intervene. But in cases where a more compact, "only the results, please" print is desired, show takes a parameter that will show only function returns: `@show.inout(only="out")`. Function calls sans returns will be show if `only='in'`.

You may find it useful that `inout` is an individually-styleable method. To highlight function entry and exit points, try:

```
show.inout.set(style='red')
```

Note: The `@show.retval` decorator has been deprecated, and will soon be removed. Please shift to `@show.inout` variants instead.

CHAPTER 9

Discovering What's There

It's often helpful to figure out "what am I dealing with here? what attributes or methods or properties are available to me?" This is where `show.dir` comes into play. You could do `show(dir(x))`, but `show.dir(x)` will provides more information, and does so more compactly. It also allows you to filter out the often huge hubbub of some objects. By default, it doesn't show any attributes starting with double underscore __. You can control what's omitted with the `omit` keyword argument. `show.dir(x, omit=None)` shows everything, while `show.dir(x, omit='_* proxy*')` omits all the methods starting with an underscore or the word "proxy."

CHAPTER 10

Interactive Limitations

As of version 1.4, `show` has good support for IPython, either running in a terminal window or in a Jupyter Notebook.

It's support for the plain interactive Python REPL is much weaker. Call it experimental. It works well at the interactive prompt, and within imported modules. It cannot, however, be used within functions and classes defined within the interactive session. This is a result of how Python supports–or rather, fails to support–introspection for interactively-defined code. Whether this is a hard limit, or something that can be worked around over time, remains to be seen. (See e.g. this discussion).

Python under Windows does not support readline the same way it is supported on Unix, Linux, and Mac OS X. Experimental support is provided for the use of `pyreadline` under Windows to correct this variance. This feature is yet untested. Works/doesn't work reports welcome!

If you want to work interactively, strongly advise you do so uner IPython not the stock REPL. Even better, in Jupyter Notebook, which is an excellent work environment in a way that the stock REPL never will be.

CHAPTER 11

API

Debugging print features.

class show.core.**NoShow**(***kwargs*)

A Show variant that shows nothing. Maintains just enough context to respond as a real Show would. Any clones will also be ‘‘NoShow‘‘s–again, to retain similarity. Designed to squelch all output in efficient way, but not requiring any changes to the source code. Maintains just enough context to

blank_lines(**args*, ***kwargs*)

Fake entry point. Does nothing, returns immediately.

changed(**args*, ***kwargs*)

Fake entry point. Does nothing, returns immediately.

clone(***kwargs*)

Create a child instance whose options are chained to this instance's options (and thence to Show.options). kwargs become the child instance's overlay options. Because of how the source code is parsed, clones must be named via simple assignment.

dir(**args*, ***kwargs*)

Fake entry point. Does nothing, returns immediately.

hr(**args*, ***kwargs*)

Fake entry point. Does nothing, returns immediately.

inout(**args*, ***kwargs*)

Fake entry point. Does nothing, returns immediately.

items(**args*, ***kwargs*)

Fake entry point. Does nothing, returns immediately.

locals(**args*, ***kwargs*)

Fake entry point. Does nothing, returns immediately.

props(**args*, ***kwargs*)

Fake entry point. Does nothing, returns immediately.

retval(**args*, ***kwargs*)

Fake entry point. Does nothing, returns immediately.

title(**args*, ***kwargs*)
Fake entry point. Does nothing, returns immediately.

class show.core.**Show**(***kwargs*)
Show objects print debug output in a 'name: value' format that is convenient for discovering what's going on as a program runs.

arg_format(*name*, *value*, *caller*, *opts*)
Format a single argument. Strings returned formatted.

arg_format_dir(*name*, *value*, *caller*, *opts*)
Format a single argument to show items of a collection.

arg_format_items(*name*, *value*, *caller*, *opts*)
Format a single argument to show items of a collection.

arg_format_props(*name*, *value*, *caller*, *opts*, *ignore_funky=True*)
Format a single argument to show properties.

call_location(*caller*)
Create a call location string indicating where a show() was called.

changed(**args*, ***kwargs*)
Show the local variables, then again only when changed.

clone(***kwargs*)
Create a child instance whose options are chained to this instance's options (and thence to Show.options). kwargs become the child instance's overlay options. Because of how the source code is parsed, clones must be named via simple assignment.

code_repr(*code*)
Return a formatted string for code. If there are any internal brace characters, they are doubled so that they are not interpreted as format template characters when the composed string is eventually output by `say`.

dir(**args*, ***kwargs*)
Show the attributes possible for the given object(s)

get_arg_tuples(*caller*, *values*)
Return a list of argument (name, value) tuples. :caller: The calling frame. :values: The with the given values.

inout(**dargs*, ***dkwargs*)
Show arguments to a function. Decorator itself may take arguments, or not. Whavevs.

items(**args*, ***kwargs*)
Show items of a collection.

locals(**args*, ***kwargs*)
Show all local vars, plus any other values mentioned.

method_push(*base_options*, *method_name*, *kwargs*)
Transitional helper function to simplify the grabbing of method-specific arguments. Will be phased out as the learnings about method arguments are piped back into Options and a more long-term API is completed.

pprint(**args*, ***kwargs*)
Show the objects as displayed by pprint. Not well integrated as yet. Just a start.

prettyprint(*mode='ansi'*, *sep=' '*, *indent=4*, *width=120*, *depth=5*, *style='friendly'*)
Convenience method to turn on pretty-printing. Mode can be text or ansi.

props(**args*, ***kwargs*)
Show properties of objects.

retval (*func*)
Decorator that shows arguments to a function, and return value, once the function is complete. Similar to inout, but only displays once function has returned.

set (***kwargs*)
Change the values of the show.

value_repr (*value*)
Return a `repr()` string for value that has any brace characters (e.g. for `dict`–and in Python 3, `set`--literals) doubled so that they are not interpreted as format template characters when the composed string is eventually output by ``say`.

where (**args*, ***kwargs*)
Show where program execution currently is. Can be used with normal output, but generally is intended as a marker.

CHAPTER 12

Notes

- `show` is in its early days. Over time, it will provide additional context-specific output helpers. For example, the "diff" views of `py.test` seem a high-value enhancement.
- `show` depends on introspection, with its various complexities and limitations. It assumes that all objects are new-style classes, and that your program has not excessively fiddled with class data. Diverge from these assumptions, and all bets are off.
- Automated multi-version testing managed with the wonderful pytest and tox. Successfully packaged for, and tested against, most late-model versions of Python: 2.7, 3.3, 3.4, 3.5, and 3.6, as well as PyPy 5.6.0 (based on 2.7.12) and PyPy3 5.5.0 (based on 3.3.5).
- The author, Jonathan Eunice or @jeunice on Twitter welcomes your comments and suggestions.

CHAPTER 13

Installation

To install or upgrade to the latest version:

```
pip install -U show
```

To `easy_install` under a specific Python version (3.4 in this example):

```
python3.4 -m easy_install --upgrade show
```

(You may need to prefix these with `sudo` to authorize installation. In environments without super-user privileges, you may want to use `pip`'s `--user` option, to install only for a single user, rather than system-wide.)

Testing

If you wish to run the module tests locally, you'll need to install `pytest` and `tox`. For full testing, you will also need `pytest-cov` and `coverage`. Then run one of these commands:

```
tox                # normal run - speed optimized
tox -e py27        # run for a specific version only (e.g. py27, py34)
tox -c toxcov.ini  # run full coverage tests
```

The provided `tox.ini` and `toxcov.ini` config files do not define a preferred package index / repository. If you want to use them with a specific (presumably local) index, the `-i` option will come in very handy:

```
tox -i INDEX_URL
```

CHAPTER 14

Change Log

1.4.7 (March 9, 2017)

Bumped test coverage to 80%. In the process, discovered and fixed some bugs with `show.props`. Most things that can be basically unit-tested, are. Largest remaining test coverage gaps concern operation under different I/O managers–esp. IPython and the standard Python REPL–that will require integration testing.

1.4.6 (March 1, 2017)

Quashed second bug related to IPython and its *%run* command, especially as used by the Enthought Canopy IDE.

1.4.5 (March 1, 2017)

Fixed problem with IPython when program run with the *%run* command. Fix esp. important for users of Enthought Canopy IDE, which uses this mode of execution extensively.

1.4.4 (February 19, 2017)

Tweak `show.prettyprint()` to not automatically multi-line all show output. If you want multi-line output, either set `show.prettyprint(sep='\m')` to globalize that preference, or use `show(..., sep='\n)` each time you want multi-line.

1.4.3 (February 2, 2017)

Bug fix: When *show.set(prefix=...)* or other settings were used, duplicate behaviors could occur, like two prefix strings printing, not just one. Also, support for Python 2.6 has been restored. Not that you should still be using that antiquated buggy. But if you are, show will once again work for you, given removal of the preventing dependency (`stuf`).

1.4.2 (January 30, 2017)

Fixed bug when location display is active (e.g. after *show.set(where=True)*) in IPython. Now correctly identifies what cell code was executed in.

1.4.0 (January 27, 2017)

Finally have good support for IPython, either in a Notebook or in a terminal/console. Suddenly, interactive use does not minimize show's usefulness (though the standard REPL still has glitches).

1.3.2 (January 26, 2017)

Fixes nasty packaging bug (failure to bundle astor sub-package into source distributions) that didn't show up in testing.

1.3.0 (January 25, 2017)

Python 3.5 and 3.6 now pass formal verification. This required embedding a 'nightly' build of astor 0.6 that has not yet made it to PyPI. Given the shift from codegen to newer astor AST-to-source library, bumping minor version.

1.2.7 (January 23, 2017)

Updated dependencies to support Python 3.5 and 3.6. These versions do not yet pass formal validation, but they do seem to work in informal testing. This is the start of a push to fully support these most recent Python implementations, and to improve support for interactive code (REPL or Jupyter Notebook).

1.2.6 (September 1, 2015)

Tweaks and testing for new version of underlying `options` module that returns operation to Python 2.6.

1.2.4 (August 26, 2015)

Major documentation reorg.

1.2.3 (August 25, 2015)

Code cleanups and additional testing. Test coverage now 77%.

1.2.1 (August 21, 2015)

Added ability to give `@show.inout` decorator its own parameters. Deprecated `@show.retval`, which is now redunant with `@show.inout(only=''out'')`.

Test coverage bumped to 71%.

1.2.0 (August 18, 2015)

Added `show.where()` as a marker for "where am I now?" Improved docs, esp. for `where`, `inout`, and `retval` methods. Improved testing. Now at 67% line coverage.

1.1.1 (August 17, 2015)

Updated testing strategy to integrate automated test coverage metrics. Immediate test and code improvements as a result. Initial coverage was 53%. Releasing now at 57%.

Clearly backed out Python 3.5 support for the moment. The AST `Call` signature has changed notably. Will need to deep-dive to fix that.

1.1.0 (August 16, 2015)

Fixed problem with underlying `say` object interactions. Some doc and testing tweaks.

1.0.4 (July 22, 2015)

Updated config, docs, and testing matrix.

1.0.2 (September 16, 2013)

Improved pretty printing of code snippets for `@show.inout` and `@show.retval` decorators.

Made `show` also accept lambdas to link to variable values.

Added `noshow` object for easy turning off of showing.

General cleanups. Tightened imports. Tweaked docs. Switched to `FmtException` from `say>=1.0.4`, and separated extensions into own module.

Drove version information into `version.py`

1.0.1 (September 9, 2013)

Moved main documentation to Sphinx format in ./docs, and hosted the long-form documentation on readthedocs.org. README.rst now an abridged version/teaser for the module.

1.0.0 (September 9, 2013)

Improved robustness for interactive use. If names cannot be detected, still gives value result with `?` pseudo-name.

Improved type names for `show.dir` and `show.props`

Improved `show.inout` with full call string on function return. A bit verbose in small tests, but too easy to lose "what was this called with??" context in real-scale usage unless there is clear indication of how the function was called.

Improved omission of probably useless display properties via `omit` keyword.

Began to add support for showing properties even when proxied through another object. Currently limited to selected SQLAlchemy and Flask proxies. More to come.

Cleaned up source for better (though still quite imperfect), PEP8 conformance

Bumped version number to 1.0 as part of move to semantic versioning, or at least enough of it so as to not screw up Python installation procedures (which don't seem to understand 0.401 is a lesser version that 0.5, because 401 > 5).

Probably several other things I've now forgotten.

Python Module Index

S

Index